

Exceptional service in the national interest



OpenACC and C++: An Application Perspective

Christian Trott

Big thanks to PGI for all the hard work

Mathew Colgrove

Brent Leback

Michael Wolfe

Douglas Miles

SAND2015-2057 C

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.



What is it about?

Goal

- add (a few) pragmas and run on GPUs
- portable
- pretty good performance

Reality end of 2013

- big performance issues since data is transferred very often
- issues with unstructured code (i.e. complex code path)
- C++!! almost didn't work
- non trivial amount of pragma decoration

“Solved” by OpenACC 2.0: Data Management Issues

Common use-case with occasional resize of data

=> cannot use simple data region around timestep loop

```
void foo(int* x, int m, int& length) {
    if(m>length) {
        delete [] x;
        x = new int[m];
        length = m;
    }
    for(int i=0;i<m;i++) {
        ...
    }
}

void run(int* x, int& length, int nsteps)
{
    for(int t = 0; t<nsteps; t++) {
        int m = bla(x,length);
        foo(x,m,length);
    }
}
```

“Solved” by OpenACC 2.0: Data Management Issues

Common use-case with occasional resize of data

=> cannot use simple data region around timestep loop

```

void foo(int* x, int m, int& length) {
    if(m>length) {
        delete [] x;
        x = new int[m];
        length = m;
    }
    for(int i=0;i<m;i++) {
        ...
    }
}

void run(int* x, int& length, int nsteps)
{
    for(int t = 0; t<nsteps; t++) {
        int m = bla(x,length);
        foo(x,m,length);
    }
}

```

Want a data region for x here, but
can't because *foo* might reallocate

“Solved” by OpenACC 2.0: Data Management Issues

Common use-case with occasional resize of data

=> cannot use simple data region around timestep loop

```
void foo(int* x, int m, int& length) {
    if(m>length) {
        delete [] x;
        x = new int[m];
        length = m;
    }
    for(int i=0;i<m;i++) {
        ...
    }
}
```

Want a data region for x here, but can't because *foo* might reallocate

```
void run(int* x, int& length, int nsteps)
{
    for(int t = 0; t<nsteps; t++) {
        int m = bla(x,length);
        foo(x,m,length);
    }
}
```

Workaround: use ‘*deviceptr*’ with ‘*acc_malloc*’ and ‘*acc_free*’

Compiler C++ issue: Class Members

- Root issue: OpenACC didn't know what to do with 'this->'

```
struct FooOpenACC {  
    double a; int N;  
    void scale(double* x) {  
        #pragma acc parallel loop copy(x[0:N])  
        for(int i=0; i<N; i++)  
            update_val(x[i]);  
    }  
    void update_val(double& x) {  
        x*=a;  
    }  
};
```

Compiler C++ issue: Class Members

- Root issue: OpenACC didn't know what to do with 'this->'

```
struct FooOpenACC {  
    double a; int N;  
    void scale(double* x) {  
        #pragma acc parallel loop copy(x[0:N])  
        for(int i=0; i<N; i++)  
            update_val(x[i]);  
    }  
    void update_val(double& x) {  
        x*=a;  
    }  
};
```

Didn't know what to do because it has implicit
'this->' in front.

Compiler C++ issue: Class Members

- Root issue: OpenACC didn't know what to do with 'this->'

```
struct FooOpenACC {  
    double a; int N;  
    void scale(double* x) {  
        #pragma acc parallel loop copy(x[0:N])  
        for(int i=0; i<N; i++)  
            update_val(x[i]);  
    }  
    void update_val(double& x) {  
        x*=a;  
    }  
};
```

Didn't know what to do because it has implicit
'this->' in front.

Compiler C++ issue: Class Members

- Root issue: OpenACC didn't know what to do with 'this->'

```
struct FooOpenACC {  
    double a; int N;  
    void scale(double* x) {  
        #pragma acc parallel loop copy(x[0:N])  
        for(int i=0; i<N; i++)  
            update_val(x[i]);  
    }  
    void update_val(double& x) {  
        x*=a;  
    }  
};
```

Didn't know what to do because it has implicit
'this->' in front.

Compiler C++ issue: Class Members

- Root issue: OpenACC didn't know what to do with 'this->'

```
struct FooOpenACC {  
    double a; int N;  
    void scale(double* x) {  
        #pragma acc parallel loop copy(x[0:N])  
        for(int i=0; i<N; i++)  
            update_val(x[i]);  
    }  
    void update_val(double& x) {  
        x*=a;  
    }  
};
```

Didn't know what to do because it has implicit
'this->' in front.

Compiler C++ issue: Class Members

- Root issue: OpenACC didn't know what to do with 'this->'

```
struct FooOpenACC {  
    double a; int N;  
    void scale(double* x) {  
        #pragma acc parallel loop copy(x[0:N])  
        for(int i=0; i<N; i++)  
            update_val(x[i]);  
    }  
    void update_val(double& x) {  
        x*=a;  
    }  
};
```

Didn't know what to do because it has implicit 'this->' in front.

Workaround: make local copies of everything

Standard C++ Issue: deep copy

When using classes it is not clear how to copy internal allocations

```
struct DataStructure {
    double* data;
    int* idx;
    int N;

    double access(int i) {
        return data[idx[i]];
    }
};

void update(DataStructure a, DataStructure b) {
    #pragma acc parallel loop
    for(int i=0; i<a.N; i++) {
        a.access(i)+=b.access(i);
    }
}
```

Standard C++ Issue: deep copy

When using classes it is not clear how to copy internal allocations

```
struct DataStructure {  
    double* data;  
    int* idx;  
    int N;
```

How to convey that 'idx' and 'data' of 'a' and 'b' have to be copied or already exist on device. How to get correct device pointers into copies of a and b.

```
};  
};
```

```
void update(DataStructure a, DataStructure b) {  
    #pragma acc parallel loop  
    for(int i=0; i<a.N; i++) {  
        a.access(i)+=b.access(i);  
    }  
}
```

Standard C++ Issue: deep copy

When using classes it is not clear how to copy internal allocations

```
struct DataStructure {  
    double* data;  
    int* idx;  
    int N;  
};
```

How to convey that 'idx' and 'data' of 'a' and 'b' have to be copied or already exist on device. How to get correct device pointers into copies of a and b.

```
void update(DataStructure a, DataStructure b) {  
    #pragma acc parallel loop  
    for(int i=0; i<a.N; i++) {  
        a.access(i)+=b.access(i);  
    }  
}
```

How that added up in miniMD 1.2

```

void ForceLJ::compute_fullneigh(Atom &atom, Neighbor &neighbor, int me)
{

    const int nlocal = atom.nlocal;
    const int nall = atom.nlocal + atom.nghost;
    const MMD_float* const restrict x = atom.d_x;
    MMD_float* const restrict f = atom.d_f;
    const int* const restrict neighbors = neighbor.d_neighbors;
    const int* const restrict numneigh = neighbor.d_numneigh;
    const int nmax = neighbor.nmax;
    const int maxneighs = neighbor.maxneighs;
    const MMD_float sigma6_ = sigma6;
    const MMD_float epsilon_ = epsilon;
    const MMD_float cutforcesq_ = cutforcesq;

#pragma acc data deviceptr(x,neighbors,numneigh,f)
{
    #pragma acc kernels
    for(int i = 0; i < nlocal; i++) {
        f[i * PAD + 0] = 0.0;
    }

    #pragma acc kernels
    for(int i = 0; i < nlocal; i++) {
        ...
    }
}

```

How that added up in miniMD 1.2

```

void ForceLJ::compute_fullneigh(Atom &atom, Neighbor &neighbor, int me)
{

  const int nlocal = atom.nlocal;
  const int nall = atom.nlocal + atom.nghost;
  const MMD_float* const restrict x = atom.d_x;
  MMD_float* const restrict f = atom.d_f;
  const int* const restrict neighbors = neighbor.d_neighbors;
  const int* const restrict numneigh = neighbor.d_numneigh;
  const int nmax = neighbor.nmax;
  const int maxneighs = neighbor.maxneighs;
  const MMD_float sigma6_ = sigma6;
  const MMD_float epsilon_ = epsilon;
  const MMD_float cutforcesq_ = cutforcesq;

#pragma acc data deviceptr(x,neighbors,numneigh,f)
{
  #pragma acc kernels
  for(int i = 0; i < nlocal; i++) {
    f[i * PAD + 0] = 0.0;
  }

  #pragma acc kernels
  for(int i = 0; i < nlocal; i++) {
    ...
  }
}

```

Extract members of other classes

How that added up in miniMD 1.2

```
void ForceLJ::compute_fullneigh(Atom &atom, Neighbor &neighbor, int me)
{
```

```
    const int nlocal = atom.nlocal;
    const int nall = atom.nlocal + atom.nghost;
    const MMD_float* const restrict x = atom.d_x;
    MMD_float* const restrict f = atom.d_f;
    const int* const restrict neighbors = neighbor.d_neighbors;
    const int* const restrict numneigh = neighbor.d_numneigh;
    const int nmax = neighbor.nmax;
    const int maxneighs = neighbor.maxneighs;
    const MMD_float sigma6_ = sigma6;
    const MMD_float epsilon_ = epsilon;
    const MMD_float cutforcesq_ = cutforcesq;
```

Extract members of other classes

```
#pragma acc data deviceptr(x,neighbors,numneigh,f)
{
    #pragma acc kernels
    for(int i = 0; i < nlocal; i++) {
        f[i * PAD + 0] = 0.0;
    }

    #pragma acc kernels
    for(int i = 0; i < nlocal; i++) {
        ...
    }
}
```

Make local copies of
members of 'this' class

How that added up in miniMD 1.2

```
void ForceLJ::compute_fullneigh(Atom &atom, Neighbor &neighbor, int me)
{
```

```
    const int nlocal = atom.nlocal;
    const int nall = atom.nlocal + atom.nghost;
    const MMD_float* const restrict x = atom.d_x;
    MMD_float* const restrict f = atom.d_f;
    const int* const restrict neighbors = neighbor.d_neighbors;
    const int* const restrict numneigh = neighbor.d_numneigh;
    const int nmax = neighbor.nmax;
    const int maxneighs = neighbor.maxneighs;
    const MMD_float sigma6_ = sigma6;
    const MMD_float epsilon_ = epsilon;
    const MMD_float cutforcesq_ = cutforcesq;
```

Extract members of other classes

```
#pragma acc data deviceptr(x,neighbors,numneigh,f)
{
    #pragma acc kernels
    for(int i = 0; i < nlocal; i++) {
        f[i * PAD + 0] = 0.0;
    }
    #pragma acc kernels
    for(int i = 0; i < nlocal; i++) {
        ...
    }
}
```

Make local copies of members of 'this' class

Use explicit device allocations (manage data movement with host and device pointers)

How that added up in miniMD 1.2

```
void ForceLJ::compute_fullneigh(Atom &atom, Neighbor &neighbor, int me)
{
```

```
    const int nlocal = atom.nlocal;
    const int nall = atom.nlocal + atom.nghost;
    const MMD_float* const restrict x = atom.d_x;
    MMD_float* const restrict f = atom.d_f;
    const int* const restrict neighbors = neighbor.d_neighbors;
    const int* const restrict numneigh = neighbor.d_numneigh;
    const int nmax = neighbor.nmax;
    const int maxneighs = neighbor.maxneighs;
    const MMD_float sigma6_ = sigma6;
    const MMD_float epsilon_ = epsilon;
    const MMD_float cutforcesq_ = cutforcesq;
```

Extract members of other classes

```
#pragma acc data deviceptr(x,neighbors,numneigh,f)
{
    #pragma acc kernels
    for(int i = 0; i < nlocal; i++) {
        f[i * PAD + 0] = 0.0;
    }
    #pragma acc kernels
    for(int i = 0; i < nlocal; i++) {
        ...
    }
}
```

Make local copies of members of 'this' class

Use explicit device allocations (manage data movement with host and device pointers)

Manually inline function calls to 'neighbor'

How that added up in miniMD 1.2

```
void ForceLJ::compute_fullneigh(Atom &atom, Neighbor &neighbor, int me)
{
```

```
    const int nlocal = atom.nlocal;
    const int nall = atom.nlocal + atom.nghost;
    const MMD_float* const restrict x = atom.d_x;
    MMD_float* const restrict f = atom.d_f;
    const int* const restrict neighbors = neighbor.neighbors;
    const int* const restrict numneigh = neighbor.numneigh;
    const int nmax = neighbor.nmax;
    const int maxneighs = neighbor.maxneighs;
    const MMD_float sigma6_ = sigma6;
    const MMD_float epsilon_ = epsilon;
    const MMD_float cutforcesq_ = cutforce
```

```
#pragma acc data deviceptr(x,neighbor.neighbors,neighbor.numneigh,f)
{
    #pragma acc kernels
    for(int i = 0; i < nlocal; i++) {
        f[i * PAD + 0] = 0.0;
    }
    #pragma acc kernels
    for(int i = 0; i < nlocal; i++) {
        ...
    }
}
```

Ex: members of other classes

Make local copies of members of 'this' class

Use explicit device allocations (manage data movement with host and device pointers)

Manually inline function calls to 'neighbor'

Working with PGI on solutions for C++

- Some of the solutions came as part of OpenACC 2.0
 - unstructured data regions, function calls
 - But: C++ still didn't work
- (i) Sandia provided stripped down examples for C++ features
- (ii) Discussed possible solutions and acceptable restrictions
- (iii) PGI fixed compiler
- (iv) Sandia tested
- (v) rinse and repeat for last 15 months

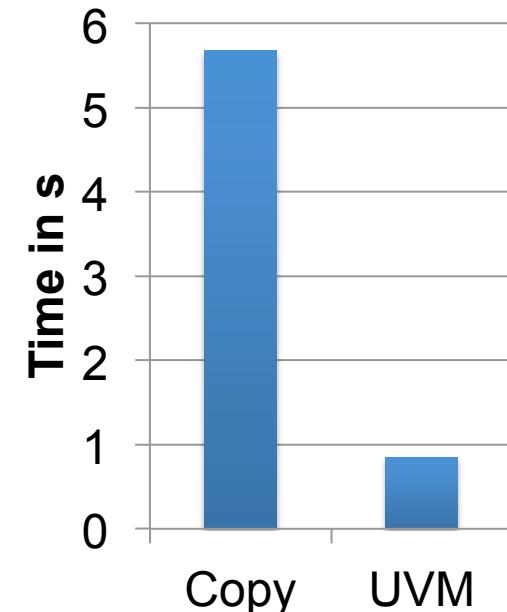
Data Management: using UVM

- Auto replace “new” allocations with UVM allocations
 - total allocations limited to device space
 - can be slower than unstructured data regions due to page size granularity
 - but effectively zero data management!!

```

int size = 10000000;
int nsteps = 100;
double* x = new double[size];

for(int k = 0; k < nsteps; k++) {
    if( k%20 != 0) {
        #pragma acc parallel loop copy(x[0:size])
        for(int i = 0; i < size; i++)
            x[i]*=a;
    } else {
        for(int i = 0; i < size; i++)
            x[i]*=a;
    }
}
  
```



Data Management: using UVM

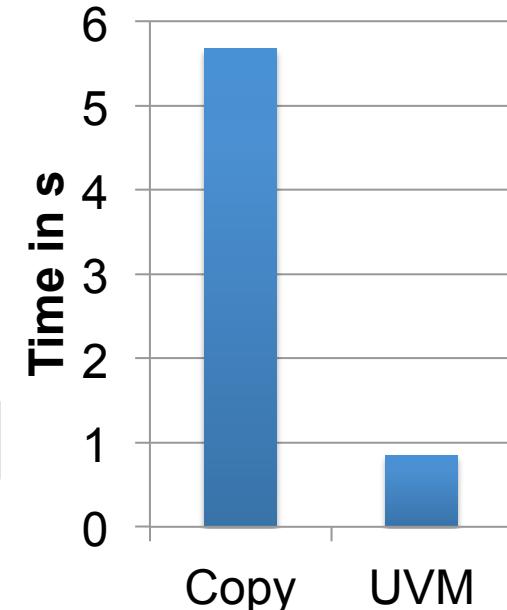
- Auto replace “new” allocations with UVM allocations
 - total allocations limited to device space
 - can be slower than unstructured data regions due to page size granularity
 - but effectively zero data management!!

```

int size = 10000000;
int nsteps = 100;
double* x = new double[size];

for(int k = 0; k < nsteps; k++) {
    if( k%20 != 0) {
        #pragma acc parallel loop copy(x[0:size])
        for(int i = 0; i < size; i++)
            x[i]*=a;
    } else {
        for(int i = 0; i < size; i++)
            x[i]*=a;
    }
}
  
```

Not Needed



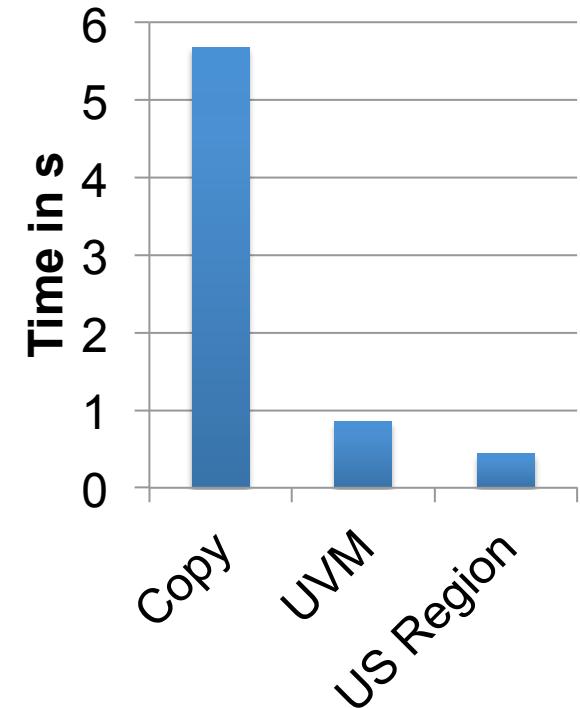
Data Management: Unstructured data regions

(And why that's sometimes better than UVM)

```

int size = 10000000;
int nsteps = 100;
double* x = new double[size];

#pragma acc enter data copyin(x[0:size])
for(int k=0; k<loop; k++) {
    if(k%20!=0) {
        #pragma acc parallel loop
        for(int i=0; i<size; i++)
            x[i]*=a;
    } else {
        #pragma acc update self(x[0:size])
        for(int i=0; i<size; i++)
            x[i]*=a;
        #pragma acc update device(x[0:size])
    }
}
#pragma acc exit data delete(x)
  
```



UVM will copy page by page: due to PCIe latency effective Bandwidth ~500MB/s

But couldn't you use structured regions?

In the previous example structured regions possible, but not with classes

```

class FooOpenACC {
    double a; double* x; int N;
public:
    FooOpenACC(double a_, double* x_, int N_):
        a(a_), x(x_), N(N_) {
        #pragma acc enter data pcreate(this)
        #pragma acc update device(this)
        #pragma acc enter data create(x[0:N])
    }
    void scale() {
        #pragma acc parallel loop
        for(int i=0; i<N; i++)
            x[i]*=a;
    }
    void host_scale() {
        update_host();
        for(int i=0; i<N; i++)
            x[i]*=a;
        update_device();
    }
    void update_host() {
        #pragma acc update self(x[0:N])
    }
    void update_device() {
        #pragma acc update device(x[0:N])
    }
};

FooOpenACC f(a,x,size);
for(int k=0;k<loop;k++) {
    if(k%20!=0) {
        f.scale();
    } else {
        f.host_scale();
    }
}

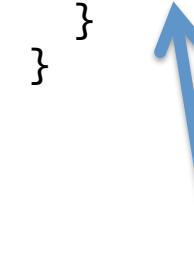
```

But couldn't you use structured regions?

In the previous example structured regions possible, but not with classes

```
class FooOpenACC {
    double a; double* x; int N;
public:
    FooOpenACC(double a_, double* x_, int N_):
        a(a_), x(x_), N(N_) {
        #pragma acc enter data pcreate(this)
        #pragma acc update device(this)
        #pragma acc enter data create(x[0:N])
    }
    void scale() {
        #pragma acc parallel loop
        for(int i=0; i<N; i++)
            x[i]*=a;
    }
    void host_scale() {
        update_host();
        for(int i=0; i<N; i++)
            x[i]*=a;
        update_device();
    }
    void update_host() {
        #pragma acc update self(x[0:N])
    }
    void update_device() {
        #pragma acc update device(x[0:N])
    }
};
```

```
FooOpenACC f(a,x,size);
for(int k=0;k<loop;k++) {
    if(k%20!=0) {
        f.scale();
    } else {
        f.host_scale();
    }
}
```



The outer loop doesn't know what FooOpenACC is doing.

Combination: C++ Classes now work for MiniMD

Unstructured data regions for device copies of class instances

```
Integrate::Integrate() {
    sort_every=20;
    #pragma acc enter data pcreate(this)
}

Integrate::~Integrate() {
    #pragma acc exit data delete(this)
}
```

Update device copy if it was potentially changed

```
void Integrate::initialIntegrate() {
    #pragma acc update device(this)
    #pragma acc parallel loop
    for(MMD_int i = 0; i < nlocal; i++) {
        v[i * PAD + 0] += dtforce * f[i * PAD + 0];
        x[i * PAD + 0] += dt * v[i * PAD + 0];
    }
}
```

Combination: C++ Classes now work for MiniMD

Unstructured data regions for device copies of class instances

```
Integrate::Integrate() {
    sort_every=20;
    #pragma acc enter data pcreate(this)
}

Integrate::~Integrate() {
    #pragma acc exit data delete(this)
}
```

Update device copy if it was potentially changed

```
void Integrate::initialIntegrate() {
    #pragma acc update device(this)
    #pragma acc parallel loop
    for(MMD_int i = 0; i < nlocal; i++) {
        v[i * PAD + 0] += dtforce * f[i * PAD + 0];
        x[i * PAD + 0] += dt * v[i * PAD + 0];
    }
}
```

Refers to device copy (e.g. device-this->...)
 Relies on UVM for x and v

Combination: C++ Classes now work for MiniMD

Unstructured data regions for device copies of class instances

```
Integrate::Integrate() {
    sort_every=20;
    #pragma acc enter data pcreate(this)
}

Integrate::~Integrate() {
    #pragma acc exit data delete(this)
}
```

Update device copy if it was potentially changed

```
void Integrate::initialIntegrate() {
    #pragma acc update device(this)
    #pragma acc parallel loop
    for(MMD_int i = 0; i < nlocal; i++) {
        v[i * PAD + 0] += dtforce * f[i * PAD + 0];
        x[i * PAD + 0] += dt * v[i * PAD + 0];
    }
}
```

Refers to device copy (e.g. device-this->...)
 Relies on UVM for x and v

Combination: C++ Classes now work for MiniMD

Unstructured data regions for device copies of class instances

```
Integrate::Integrate() {
    sort_every=20;
    #pragma acc enter data pcreate(this)
}
```

```
Integrate::~Integrate() {
    #pragma acc exit data delete(this)
}
```

Update device copy if it was potentially changed

```
void Integrate::initialIntegrate() {
    #pragma acc update device(this)
    #pragma acc parallel loop
    for(MMD_int i = 0; i < nlocal; i++) {
        v[i * PAD + 0] += dtforce * f[i * PAD + 0];
        x[i * PAD + 0] += dt * v[i * PAD + 0];
    }
}
```

Refers to device copy (e.g. device-this->...)
 Relies on UVM for x and v

Combination: C++ Classes now work for MiniMD

Unstructured data regions for device copies of class instances

```
Integrate::Integrate() {
    sort_every=20;
    #pragma acc enter data pcreate(this)
}
```

```
Integrate::~Integrate() {
    #pragma acc exit data delete(this)
}
```

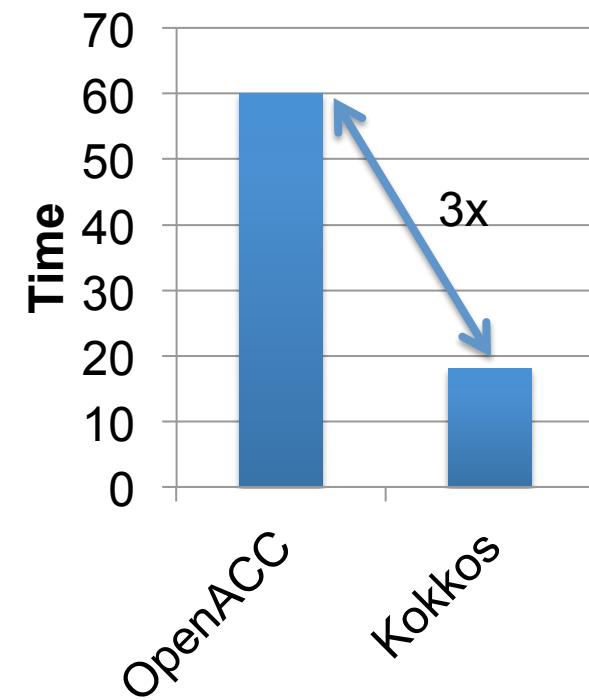
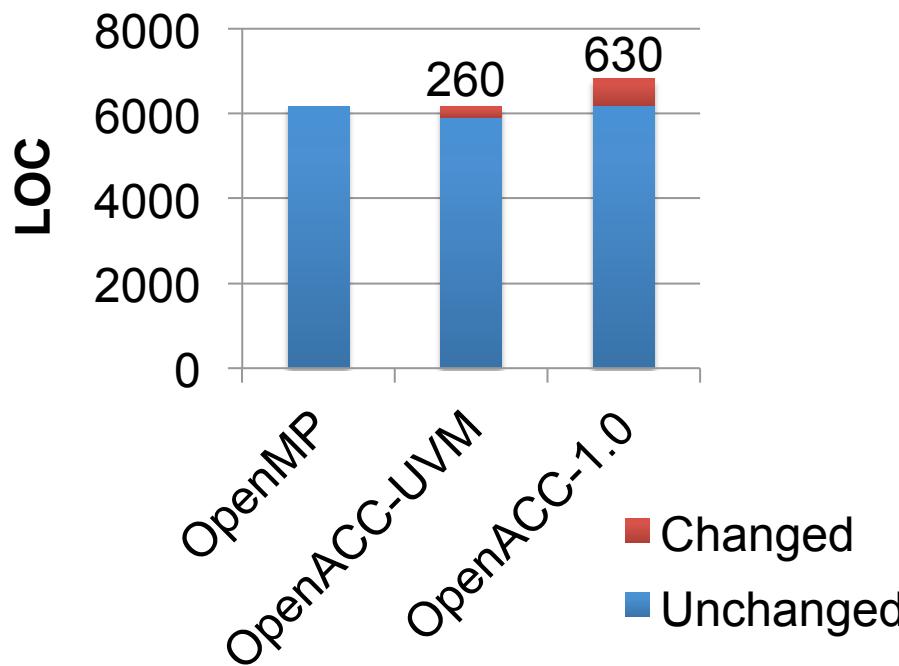
Update device copy if it was potentially changed

```
void Integrate::initialIntegrate() {
    #pragma acc update device(this)
    #pragma acc parallel loop
    for(MMD_int i = 0; i < nlocal; i++) {
        v[i * PAD + 0] += dtforce * f[i * PAD + 0];
        x[i * PAD + 0] += dt * v[i * PAD + 0];
    }
}
```

Refers to device copy (e.g. device-this->...)
 Relies on UVM for x and v

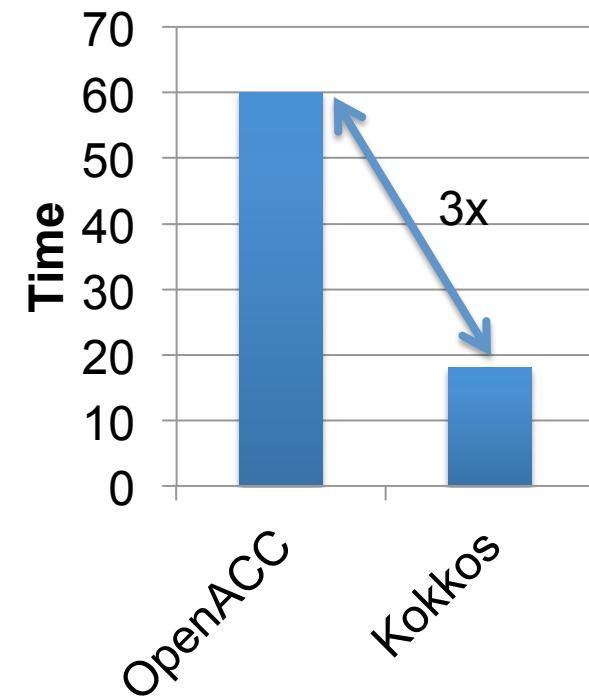
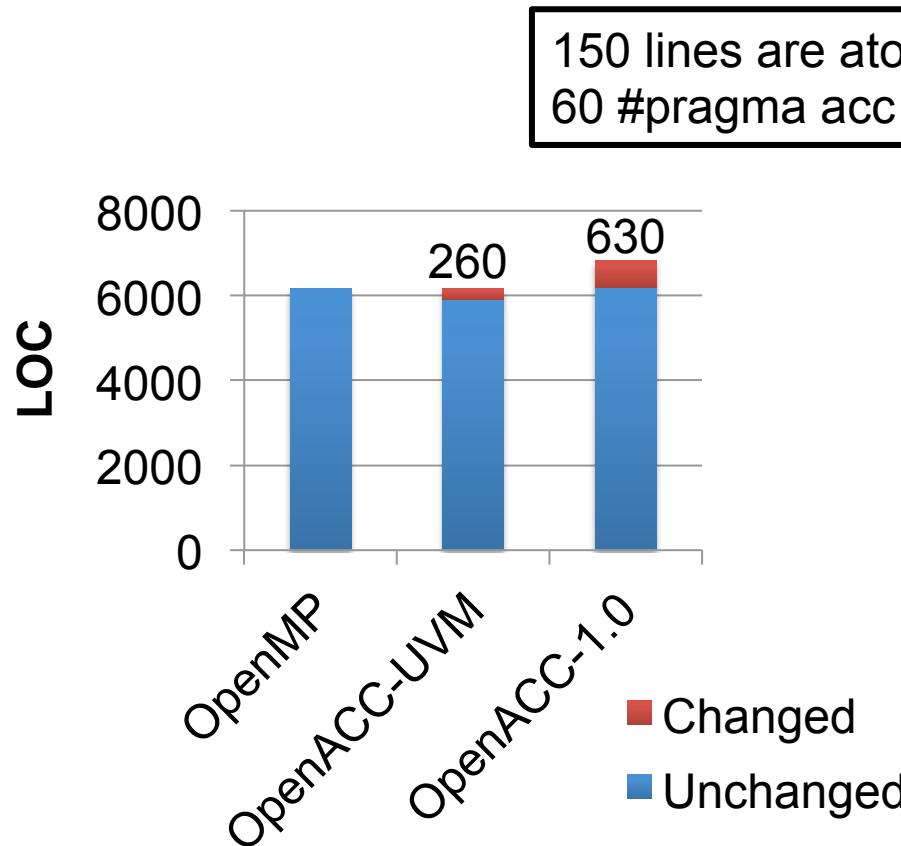
miniMD-OpenACC-UVM 2.0

- based on miniMD-OpenMP-Scalable => no algorithm changes
- use UVM to manage data transfers => minimal data clauses
- biggest change: atomic counters need to be allocated



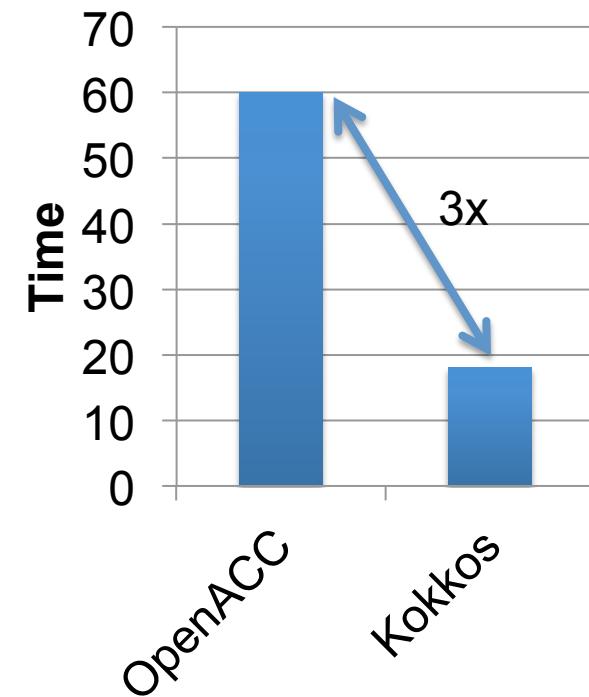
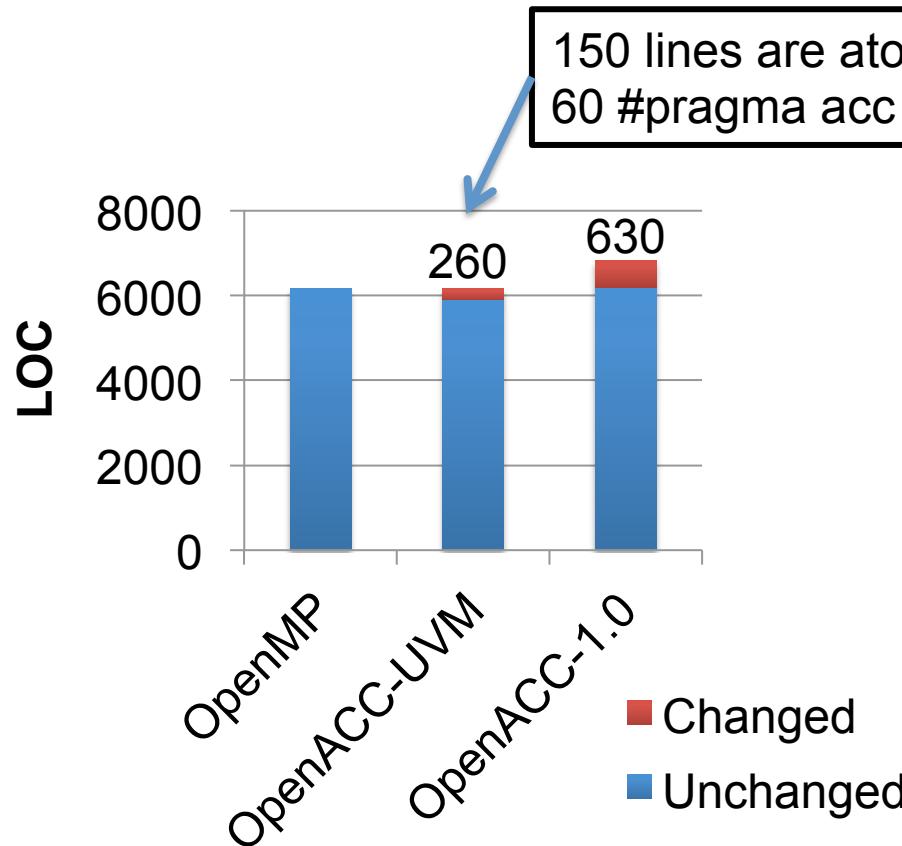
miniMD-OpenACC-UVM 2.0

- based on miniMD-OpenMP-Scalable => no algorithm changes
- use UVM to manage data transfers => minimal data clauses
- biggest change: atomic counters need to be allocated



miniMD-OpenACC-UVM 2.0

- based on miniMD-OpenMP-Scalable => no algorithm changes
- use UVM to manage data transfers => minimal data clauses
- biggest change: atomic counters need to be allocated



Open Issues

- Still many bugs in the compiler: please try out and report (turnaround and responsiveness is pretty good)
- Compiler vendor specific solutions: what about generic concept of UVM, implicit routine generation?
- Sometimes tricky to get the order right (class creation / attach) and to know when routine is needed and when not
- Performance issues compared with Cuda
 - classes are copied into global memory, much slower than constant cache
 - its hard to get the settings right to get `__ldg` loads
 - `__ldg` loads are slower than `texfetch1D` in some cases

Not yet ready for prime time, but it is getting there.



Sandia
National
Laboratories

Exceptional service in the national interest

Questions and further discussion: crtrott@sandia.gov